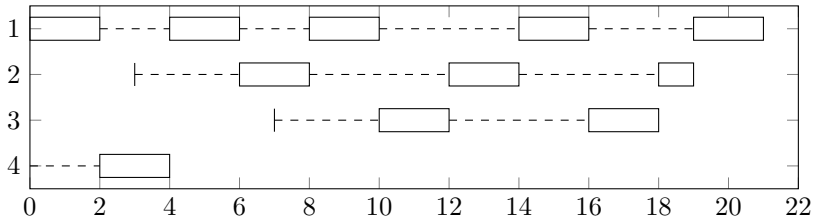
1. (a) Med *round-robin* schemaläggs processerna som följer:



Ready-kön har följande innehåll. Processen markerad med en asterisk exekeveras. Parenteser anger kvarvarande exekeveringstid.

Tick	**Innehåll**
0	T1(10)*, T4(2)
2	T4(2)*, T1(8)
3	T4(1)*, T1(8), T2(5)
4	T1(8)*, T2(5)
6	T2(5)*, T1(6)
7	T2(4)*, T1(6), T3(4)
8	T1(6)*, T3(4), T2(3)
10	T3(4)*, T2(3), T1(4)
12	T2(3)*, T1(4), T3(2)
14	T1(4)*, T3(2), T2(1)
16	T3(2)*, T2(1), T1(2)
18	T2(1), T1(2)
19	T1(2)
21	-

(b) Väntetiden kan exempelvis läsas ut ifrån grafen i (a) genom att räkna längden på de sträckade linjerna. Alternativt, sluttid - starttid - längd på jobbet.

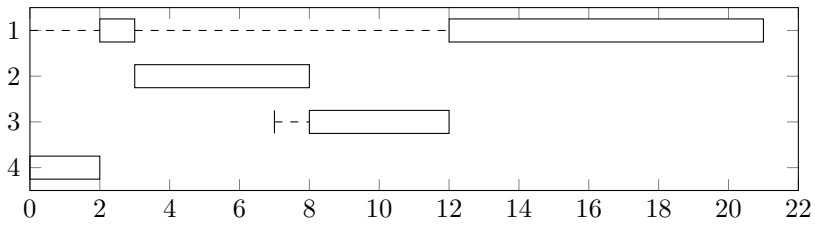
P1 21 - 0 - 10 = 11

P2 19 - 3 - 5 = 11

P3 18 - 7 - 4 = 7

P4 4 - 0 - 2 = 2

(c) Med prioritetsbaserad schemaläggning blir det som nedan. Notera att ordningen i ready-kön inte spelar någon roll, endast prioriteten spelar roll.



Ready-kön har följande innehåll. Processen markerad med en asterisk exekeveras. Parenteser anger prioritet samt kvarvarande exekeveringstid.

Tick	**Innehåll**
0	T1(1, 10), T4(5, 2)*
2	T1(1, 10)*
3	T1(1, 9), T2(4, 5)*
7	T1(1, 9), T2(4, 1)*, T3(3, 4)
8	T1(1, 9), T3(3, 4)*
12	T1(1, 9)*
21	-

(d) Väntetiden kan beräknas likt i (b), och blir:

P1 21 - 0 - 10 = 11

P2 8 - 3 - 5 = 0

P3 12 - 7 - 4 = 1

P4 2 - 0 - 2 = 0

2. (a) Starvation kan uppstå i prioritetsbaserad schemaläggning. För att illustrera problemet, antag att vi har ett system med en tråd, T1, som har prioritet 1. Det finns också två trådar, T2 och T3, som båda har prioritet 4. Trådarna T2 och T3 väntar på data från disken. När de får sin data så kör de i 4 tidsenheter och frågar efter ny data från disken. Både T2 och T3 gör detta i en oändlig loop. Om disken är tillräckligt snabb och levererar data till T2 och T3 på under 4 tidsenheter, så kommer T2 och T3 kunna fylla ut CPU-tiden, och T1 får aldrig någon tid att köra.

Exemplet ovan är ekvivalent med att det var 4:e tidsenhet startas en ny tråd med prioritet 4 och en körtid av 4 (kom ihåg: vi schemalägger s.k. *CPU-bursts*).

(b) Starvation kan *inte* uppstå i *round-robin* som den är beskriven. I och med att nya processer alltid läggs till i slutet av ready-kön och inte avbryter tråden som körs, så är det inte möjligt för trådar att ”tränga sig” i ready-kön. Tillsammans med ett tidskvantum på 2 ticks gör detta att alla trådar kommer få köra i sinom tid, och ingen tråd riskerar att behöva vänta oändligt länge.

3. Vår disk är 2^{42} bytes, och har block som är 2^{12} bytes stora.

   (a) Totalt finns det $2^{42}/2^{12} = 2^{30}$ block på disken. Vi behöver alltså 30 bitar för att lagra ett blocknummer. För att lagra dessa 30 bitar behöver vi 4 bytes.

   (b) Givet 4 bytes för att lagra blocknummer, så kommer vår FAT bli $2^{30} \cdot 2^2 = 2^{32}$ bytes stor, eller 4 GiB.

   (c) Om vi använder indexerad allokering kommer varje indexblock kunna innehålla $2^{12}/2^2 = 2^{10}$ pekare till andra block. Första nivån innehåller alltså 2^{10} pekare till andra indexblock, som i sin tur innehåller 2^{10} pekare till block med data. Totalt kan en fil alltså innehålla $2^{10} \cdot 2^{10}$ block, vilket motsvarar $2^{10} \cdot 2^{10} \cdot 2^{12} = 2^{32}$ bytes, eller 4 GiB.

   (d) För att både kunna lagra stora filer och minska overhead för mindre filer kan vi använda en approach liknande den som används i inoder i UNIX. I vårt första indexblock kan vi exempelvis låta de första 2^9 pekarna peka direkt på datablock (dvs. som i indexering med en nivå) för att minska overhead för små filer. Nästa 2^8 pekare kan peka på indexblock (som vardera innehåller 2^{10} pekare som tidigare). De sista 2^8 pekarna kan sedan peka på indexblock som har dubbel indirektion (dvs. de pekar på indexblock som vardera innehåller 2^{10} pekare till indexblock med 2^{10} pekare vardera).

   Detta leder till en maximal filstorlek på:

$$
\begin{aligned}
2^9 \cdot 2^{12} + & \quad \text{direkta pekare} \\
2^8 \cdot 2^{10} \cdot 2^{12} + & \quad \text{1 nivå indirektion} \\
2^8 \cdot 2^{10} \cdot 2^{10} \cdot 2^{12} = & \quad \text{2 nivåer indirektion} \\
2^{21} + 2^{30} + 2^{40} > 1 \text{ TiB} &
\end{aligned}
$$

4. (a) Adressen består i det här fallet av ett page-nummer (`0x8`) och en offset (`0x31`). Page-numret slås upp i page-tabellen för respektive process. För process 1 får vi frame-nummer `0xA3`, och för process 2 får vi frame-nummer `0x00`. Vi noterar också att valid-biten är 1 i båda fallen. Vi kan därmed forma den fysiska adressen genom att slå ihop frame-numret med den offset som fanns i originaladressen. För process 1 får vi `0xA331` och för process 2 får vi `0x0031`.

   (b) Ja, processerna delar minne. Om vi jämför page-tabellerna med varandra ser vi att det finns två frames som förekommer i båda page-tabellerna: `0x10` (rad `0x0` resp. `0x1`) och `0x31` (rad `0xB` resp. `0x4`). Vi noterar dock att valid-biten inte är satt på rad `0x0` för process 1, vilket innebär att process 1 inte kan komma åt frame `0x10`. Alltså delas bara frame `0x31` mellan processerna. Process 1 kommer åt den delade frame:n via `0xB00` till `0xBFF`, och process 2 via `0x400` till `0x4FF`. Alltså är refererar adressen `0xB2F` i process 1 till samma minne som adressen `0x42F` i process 2.

   (c)

$$
\begin{aligned}
\text{EAT} &= 0.8 \cdot 10 \text{ ns} + 0.2 \cdot 2 \cdot 10 \text{ ns} \\
&= 8 \text{ ns} + 4 \text{ ns} \\
&= 12 \text{ ns}
\end{aligned}
$$

5. (a) För att minimera svinn i systemet vill vi hitta värden på page-storlek och en uppdelning av virtuella adresser som gör att våra page-tabeller är så nära storleken på en page som möjligt.

   Om vi väljer en page-storlek på 16 KiB får vi en offset på 14 bitar. Detta ger oss att ett frame-nummer behöver vara $40 - 14 = 28$ bitar stort. Med 16 KiB pages så kan vi alltså klara oss med en page-tabell med rader som är 4 bytes stora.

   Givet detta så kan vi komma fram till att varje page-tabell kan innehålla $2^{14}/2^2 = 2^{12}$ element. Detta gör att varje nivå av page-tabell kan ”konsumera” 12 bitar.

   Med allt detta kan vi då dela upp virtuella adresser i 12 + 12 + 14 bitar – 12 bitar för första nivån, 12 bitar för andra nivån och 14 bitar offset.

   (b) Givet designen ovan så kan vi adressera $2^{12} \cdot 2^{14} = 2^{26}$ bytes från varje pagetabell i den andra nivån. Då detta är mer än en megabyte behöver vi endast en page-tabell för första nivån och en page-tabell för andra nivån. Detta ger oss totalt $2 \cdot 16$ KiB = 32 KiB.

6. (a) För att se till så att bara anställda ska komma åt mappen `/exams` så sätter vi gruppen `employee` som grupp på mappen, och ger den rättigheterna `rwxrwx---`. Ägare till mappen är inte viktigt, men vi kan anta att den ägs av exempelvis `root`.

   Mapparna under `/exams` kan hanteras på liknande sätt: dvs. de ska tillhöra gruppen `employee` så att olika examinatorer kan skapa filer i mappen, rättigheterna ska vara `rwxrwx---`, och ägare spelar ingen stor roll och kan vara exempelvis `root`. I praktiken kommer dock ägaren vara den som skapade mappen (en examinator, eller någon från tentaservice).

   För individuella filer behöver vi vara lite mer försiktiga. Varje tenta-PDF kommer ägas av examinatorn som skapade filen. Gruppen för filen måste vara `exam` så att tentaservice kan läsa den. Till sist måste rättigheterna vara: `rw-r-----` så att endast tentaservice kan läsa filen.

   Notera: här antas att alla i gruppen `exam` också är medlemmar i gruppen `employees`, i och med att personal på tentaservice också är anställda vid universitetet.

   (b) I och med att vi inte har möjlighet att schemalägga jobb på systemet så måste vi använda oss av ett SETUID-program. Programmet kan exempelvis ta emot ett datum och en kurskod som kommandoradsargument. Programmet måste sedan kontrollera att datumet inte är dagens datum eller i framtiden, och kan sedan kopiera den efterfrågade filen till en lämplig plats med rättigheter lämpade för den som startade programmet.

   För att detta ska fungera så måste filen som innehåller programmet ägas av en användare som har tillgång till filerna i filsystemet. Ett exempel är att låta programmet ägas av `root`. Ett bättre alternativ är dock att låta programmet ägas av en användare vi kallar `exam-get`, som är medlem i gruppen `exam`. Detta gör att programmet kan få privilegierna av de som är i `exam`-gruppen. Detta gör att programmet får tillräckligt med rättigheter för att göra sitt jobb, utan att behöva vara `root`.

   (c) Den viktigaste mekanismen är dual-mode-exekevering. Detta gör att program som körs i user-mode inte kan kommunicera direkt med hårdvaran. Program måste därmed be

operativsystemet om hjälp med att läsa filerna, vilket i sin tur gör att operativsystemet kan kontrollera rättigheter för filerna.